Osiris: A Tool for Abstraction and Verification of Control Software with Lookup Tables

Nikos Aréchiga
Toyota InfoTechnology Center, USA
465 Bernardo Avenue
Mountain View, California 94043
narechiga@us.toyota-itc.com

Sumanth Dathathri
California Institute of Technology
1200 E California Blvd
Pasadena, CA 91125
sdathath@caltech.edu

Shashank Vernekar
Toyota InfoTechnology Center, USA
465 Bernardo Avenue
Mountain View, California 94043
svernekar@us.toyota-itc.com

Nagesh Kathare
Toyota InfoTechnology Center, USA
465 Bernardo Avenue
Mountain View, California 94043
nkathare@us.toyota-itc.com

Sicun Gao
Massachusetts Institute of Technology
77 Massachusetts Avenue
Cambridge, MA 02139
sicung@csail.mit.edu

Shinichi Shiraishi
Toyota InfoTechnology Center, USA
465 Bernardo Avenue
Mountain View, California 94043
sshiraishi@us.toyota-itc.com

ABSTRACT

Some industrial systems are difficult to formally verify due to their large scale. In particular, the widespread use of lookup tables in embedded systems across diverse industries, such as aeronautics and automotive systems, create a critical obstacle to the scalability of formal verification. This paper presents Osiris, a tool that automatically computes abstractions of lookup tables. Osiris uses these abstractions to verify a property in first order logic. If the verification fails, Osiris uses a falsification heuristic to search for a violation of the specification. We validate our technique on a public benchmark of an adaptive cruise controller with lookup tables.

CCS CONCEPTS

•Software and its engineering →Software verification;

KEYWORDS

Abstraction, Lookup Tables, Verification, Formal Methods

ACM Reference format:
Nikos Aréchiga, Sumanth Dathathri, Shashank Vernekar, Nagesh Kathare, Sicun Gao, and Shinichi Shiraishi. 2017. Osiris: A Tool for Abstraction and Verification of Control Software with Lookup Tables. In *Proceedings of Safe Control of Connected and Autonomous Vehicles, Pittsburgh, Pennsylvania USA, April 2017 (SCAV'17),* 8 pages.
DOI: http://dx.doi.org/10.1145/3055378.3055384

1 INTRODUCTION

Cyberphysical systems are growing in scale and complexity, and are being deployed for a variety of applications. Some of these applications are safety-critical, such as aeronautics and driver-assistance features, and others require high performance and quality of service, such as network systems. These applications require the highest level of assurance. Existing formal verification techniques, however, are difficult to scale to many of these applications. For example, a component in a production system that we have considered contains over 10^{50} total proof cases, which would require an estimated 10^{34} years on a machine with one million cores, assuming 0.01 seconds per proof case.

Based on our experience with industrial systems, we have observed that the widespread use of lookup tables creates a critical obstacle to scalability of formal methods. Lookup tables are an important and irreplaceable element of modern engineering design.

Some lookup tables are used to model severely nonlinear physical components—for which no satisfactory equational model exists. Others serve as control laws in cases where no traditional control design method delivers the required performance. Others still serve as arithmetic shortcuts to quickly compute complex functions, such as trigonometrics, in embedded systems with timing constraints.

Lookup-tables challenge traditional verification techniques because each entry of the lookup table must be treated as a separate case. If the system under analysis contains a large number of cascaded lookup tables, the number of proof cases grows exponentially, quickly outstripping the ability of a supercomputer to deliver timely verification results as part of a product-development cycle.

We have developed a tool, Osiris, which reconstructs an approximation of the function implemented by the lookup table, and uses this approximation to construct upper and lower bounding functions on the lookup table data. Osiris uses an SMT solver as a back-end to verify that the lookup table data is correctly abstracted, and then uses the abstractions to prove the desired property.

If the proof attempt fails, Osiris extracts a *candidate counterexample* or a specific case in which the abstractions hold but the desired specification is violated. This candidate counterexample can be used to evaluate situations in which the system may nearly violate its specification, and can be used to search for a true counterexample. If a true counterexample cannot be found, Osiris refines its abstractions by increasing the degree of the abstractions, i.e. moving from linear to quadratic, and then to cubic.

We illustrate the performance of our approach on a cruise control benchmark published by Toyota InfoTechnology Center. This benchmark consists of a controller with a monitor that tries to detect dangerous conditions [16]. This benchmark contains three lookup tables. The simplest lookup table is one-dimensional and the

most complex lookup table is three-dimensional. The total number of combinations of lookup table outputs is 77,409,024.

The paper is structured as follows. Section 2 describes related work, Section 3 provides background on lookup tables and SMT verification approaches, and Section 4 explains our problem statement. We defer a detailed treatment of Osiris' abstraction computation technique to a future publication, but Section 5 describes a high-level view of this procedure, as well as how we use them for verification and falsification, and how we refine them if the specification cannot be proved nor falsified. Section 6 describes Osiris, Section 7 presents our case study, and Sections 8 and 9 conclude and describe directions for future work.

2 RELATED WORK

To the best of our knowledge, research interest in formal verification of lookup tables is recent. The authors of [13] opine that abstractions should be valuable for analysis of large models with lookup tables, but do not develop a concrete abstraction scheme. Our present work develops a tool to automatically compute abstractions via a counterexample-guided abstraction refinement method, a common paradigm to iteratively generate and improve abstractions [4]

The work of [12] uses a user-assisted mechanical theorem prover to prove safety of a large-scale aircraft collision avoidance system, which includes a large lookup table. The work of [12], however, relies on a human user to provide insight and manually reduce the system to simpler forms, until it is possible to derive input-output conditions on the lookup table to guarantee safety. This technique works top-down, starting from an overall system specification and decomposed with user assistance to a specification on the lookup table itself. This technique would be difficult to apply in a scenario with multiple cascaded lookup tables, since it would be difficult to decompose the high-level specification into obligations for each table, which would require the computationally infeasible task of propagating logical formulas through the tables. In contrast, our technique works bottom-up, treating the lookup table as training data for an automatic learning procedure, which learns an abstraction of the lookup table. This abstraction is then used as part of an SMT query to check that the system specification is satisfied.

The work of [11] uses a technique to eliminate proof cases that are not reachable, and then analyzing the remaining cases in parallel with the theorem prover PVS. [11] run their technique on the same public benchmark as we do, and reduce the approximately seven million proof cases to approximately seventy thousand for a runtime of approximately four hours. In contrast, our tool uses an SMT backend instead of a theorem prover. The choice of an SMT tool choice affords greater automation and greater speed. Our abstraction-based tool can handle this benchmark in 30 seconds on a machine with the same specifications (44 cores, 256 GB RAM). Further, we believe that our abstraction technique can better scale to larger models, since our abstractions decompose the system and prevent the case explosion to begin with, instead of filtering a large set of cases. This greater speed, however, comes at a price: if the proof fails, there is no partial proof that a designer can analyze. To address this issue, Osiris also implements a falsification heuristic to automatically search for error traces.

3 BACKGROUND

3.1 Lookup Tables

Informally, a lookup table is a function defined by a table of input and output values. A lookup table maps certain points of its input space, called *breakpoints*, to values prescribed by a given table, such as the one shown in Table 1. Note that despite the tabular structure, Table 1 represents an n-dimensional lookup table, *not* a two-dimensional one. The output of the function for values that do not appear in the table are computed by some given interpolation function if they are contained in the range of the breakpoints, and by some extrapolation function otherwise.

$x_1^{(1)}$	...	$x_i^{(1)}$	...	$x_n^{(1)}$	$y^{(1)}$
$\vdots$		$\vdots$		$\vdots$	$\vdots$
$x_1^{(j)}$	...	$x_i^{(j)}$	...	$x_n^{(j)}$	$y^{(j)}$
$\vdots$		$\vdots$		$\vdots$	$\vdots$
$x_1^{(m)}$	...	$x_i^{(m)}$	...	$x_n^{(m)}$	$y^{(m)}$

Table 1: Lookup Table with n inputs and m breakpoints

Formally, an n-dimensional lookup table with m-breakpoints is a function $\lambda : \mathbb{R}^n \to \mathbb{R}$, such that

(1) for each breakpoint $(x^{(k)}, y^{(k)})$ ($k \in \{1, \ldots, m\}$) that appears in the table, $\lambda(x^{(k)}) = y^{(k)}$, and
(2) for every point $x \in \mathbb{R}^n$ that does not appear in the table,
   (a) if each component x_i is contained in the range of the lookup table, i.e. $\min_k(x_i^{(k)}) \leq x_i \leq \max_k(x_i^{(k)})$ for each $i \in \{1, \ldots, n\}$, then $\lambda(x)$ is given by some interpolation function `interp`.
   (b) otherwise, $\lambda(x)$ is given by some extrapolation scheme `extrap`.

Our approach is general, and can be applied to any interpolation and extrapolation functions. However, in our case study, we will interpolate the lookup table by the multilinear interpolation formula described in [3]. For n dimensions, we will use the notation

$$\texttt{multiLinInterp}_n((x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), x)$$

to mean the n-dimensional interpolation function between points $(x^{(1)}, y^{(1)})$ and $(x^{(2)}, y^{(2)})$, evaluated at x. For simplicity, we will not extrapolate the lookup tables in our case study and simply assume that the range of interest is restricted to the range of the lookup tables.

3.2 Lookup tables as logical formulas

Our technique relies on the ability to encode the system and its specification into first-order logic. An n-dimensional, m-breakpoint lookup table can be encoded as a first-order logical formula as follows.

Consider, without loss of generality, a two-dimensional lookup table with m breakpoints. The k-th breakpoint can be encoded by

the following logical formula when $k = 1, \ldots, m-1$.

$$b_k \equiv x_1^{(k)} \leq x_1 \leq x_1^{(k+1)} \ \wedge \ x_2^{(k)} \leq x_2 \leq x_2^{(k+1)} \rightarrow$$
$$y = \texttt{multiLinInterp}_2((x^{(k)}, y^{(k)}), (x^{(k+1)}, y^{(k+1)}), x)$$

The vector x is the input of the lookup table, and x_1 and x_2 are its components. The function $\texttt{multiLinInterp}_2$ is bilinear interpolation. Similar expressions can be derived for lookup tables of other dimensions.

The overall lookup table can be expressed by the conjunction of the logical formulas for the breakpoints.

$$L \equiv \bigwedge_{k=1}^{m} b_k \tag{1}$$

3.3 Satisfiability Modulo Theories

Let $\Gamma(x)$ be a set of logical formulas with a vector of free variables x, and suppose x takes values in $\mathbb{R}^n$. The problem of *satisfiability modulo theory of the reals* is to find a point $r \in \mathbb{R}^n$ such that the logical formula $\Gamma(r)$ is true, or prove that none exists. In this case we say that r *satisfies* Γ.

Solvers exist that can solve the problem of satisfiability modulo theory of the reals for *subsets* of first-order logic, such as logical constraints that contain only polynomial functions over the reals [5]. The general problem is referred to as *satisfiability modulo theories* (SMT), since these solvers frequently support logical formulas over other sets, such as naturals or floating point numbers. Other solvers support transcendental functions, but relax the problem to finding an approximate solution or proving that not even an approximate solution exists [8].

4 PROBLEM FORMULATION

In this paper, we consider the problem of proving input-output properties of controllers. For future work, we plan to extend our technique closed-loop properties of a control system, by applying our abstraction technique to the lookup tables, and then using a specialized solver to reason about the continuous portion of the dynamics. For example, bounded-time properties could be handled by a tool such as [6, 7, 9]. Unbounded-time properties would need to be supplemented by an automated invariant-guessing heuristic, such as [14].

Our approach can be applied to controller models that are given in first-order logic. In industry, controllers are frequently modeled either as imperative programs (for example in C), or as signal-flow models, e.g. Simulink.

In the case of imperative languages, Dynamic Logic provides a generic framework for translating common imperative control structures into first-order logic [10].

Numerous semantics have been proposed for signal-flow languages such as Simulink [2, 15] and Lustre. For our purposes, it suffices that the selected semantics should result in first-order logical constraints. We assume one of these existing semantic interpretations has already been used to translate the model appropriately, and in our case study we perform the translation manually.

Regardless of the original format of the controller, we assume that it has been translated to a set of logical constraints $\Sigma(x)$, not including any lookup tables, where x is the vector of *all* variables that occur in the system, including inputs, outputs, and intermediate assignment variables. We handle the lookup tables separately, and assume that each lookup table, indexed by i has been encoded as the first-order logic formula $L_i(x)$ as described in Section 3. Similarly, we assume that the specification is given as a first-order formula $S(x)$.

Then, the problem is to determine whether there exists a value of the variables x that

(1) satisfies the model constraints $\Sigma(x)$, i.e., the values are related to each other according to the structure of the model;
(2) satisfies each L_i, i.e., the values are related to each other in a way that satisfies the mapping produced by the lookup tables; and
(3) does not satisfy the specification $S(x)$, i.e., it is an erroneous condition.

To check for the existence of this kind of erroneous condition, we can use an SMT solver to check the satisfiability of the following logical formula, assuming the number of lookup tables in the model is N.

$$\left(\bigwedge_{i=1}^{N} L_i(x) \right) \wedge \Sigma(x) \wedge \neg S(x)$$

This logical formula states that values for the vector of variables x must satisfy each lookup table L_i as well as the model constraints $\Sigma(x)$. In addition, the value x should *falsify* the safety condition $S(x)$. If no such value exists, then the system is guaranteed to be defect-free.

The key obstacle to directly checking this condition is that the lookup table formulas L_i are large, and this large scale is difficult. Further compounding this problem, each entry of the lookup table is encoded as an implication, which induces a case analysis: each range on the left-hand side of the implication is a case, and the right-hand side of the implication is the value of the table at that case. If we assume for simplicity that all lookup tables have m cases, and that there are k lookup tables in a model, hence the total number of cases is m^k. This exponential explosion in cases forbids a naive analysis.

Our approach is to generate an abstraction A_i for each L_i by using the lookup table data as training data to learn parameters in an abstraction template. As a result, the logical formula will be simplified, but the abstraction loses information. To address this, we provide a falsification heuristic that can help to find true counterexamples when the verification does not succeed.

5 COMPUTING ABSTRACTIONS

We will defer a thorough treatment of our abstraction computation technique for a future publication, so we will only give a high-level overview here. Our approach to improve scalability is to abstract the lookup tables by *functional intervals*. A functional interval is a function that for each argument $x \in \mathbb{R}^n$ returns a (closed) interval over R, $A(x) = [a(x), b(x)]$ where $a(x)$ is the lower bounding envelope around the lookup table and $b(x)$ is the upper bounding envelope. We say that a functional interval $A(x)$ *abstracts* a lookup table $L(x)$ over a set $S \subseteq \mathbb{R}^n$ if for every $x \in S$, $L(x) \in A(x)$.

A functional interval abstraction is an *overapproximation* of a lookup table, in the sense that a property that holds for all values

in the interval $A(x)$ must also hold of $L(x)$, but not vice-versa. The abstraction loses precision, but provides a simplification if the functions $a(x)$ and $b(x)$ have a sufficiently simple structure.

As a result, a procedure to compute a functional interval abstraction must balance between two conflicting requirements. On the one hand, it should be as precise as possible, by keeping the size of the interval small for every x, but it must also have a simple arithmetic structure, preferably consisting of linear or low-order polynomial terms, so that proving that the desired property holds of the abstraction is as simple as possible.

To navigate these conflicting requirements, we first try to abstract the lookup tables with linear abstractions, and see if these simple abstractions are sufficient to prove the specification or to guide the search to a counterexample. If the simple, linear abstractions are insufficient, then we iteratively increase the complexity to a quadratic template, then to cubic, etc. As we describe in Section 6, our tool uses a library of abstraction templates that are indexed by complexity, and iterates through them on each subsequent abstraction attempt.

In the following, we will describe our procedure for computing abstractions from approximations, and how these same abstractions can guide the search for a counterexample when the specification cannot be proved in the first attempt.

5.1 Computing abstractions by approximation

We use a regression-based procedure to automatically compute a functional interval for each lookup table in the model. First, we fix a parametric template for a function that approximates the lookup table data, and then we will proceed to learn parameter values that allow the function to approximate the lookup table data. Next, we use bisection search to search for the smallest offset that can be added and subtracted from the approximation to yield upper and lower bounds for the lookup table function.

We begin by computing an approximation of the lookup table data. Formally, let $f(a,x)$ be a function parametrized by $a \in \mathbb{R}^p$, with the same domain and range as the lookup table function L. We solve a regression problem to find the value of the parameter vector a that minimizes the mean-squared error over the breakpoints of the lookup table.

$$\underset{a}{\text{minimize}} \sum_{i=1}^{k} (y^{(k)} - f(a, x^{(k)}))^2$$

Let a^* be the value of a found by this optimization problem. Next, we use the approximation $f(a^*,x)$ to find a functional interval. We begin by setting the offset to some initial value, eg. $\epsilon = 1$. Then, we use an SMT solver to check whether the lower and upper offset functions $f(a^*,x) - \epsilon$ and $f(a^*,x) + \epsilon$ are lower and upper bounds for the lookup table function over all values in the range of interest $S \subseteq \mathbb{R}^n$. This is equivalent to checking the validity of the following logical formula with an SMT solver.

$$\forall x \in R \,.\, f(a^*,x) - \epsilon \leq L(x) \;\wedge\; L(x) \leq f(a^*,x) + \epsilon$$

Note that the expression for $L(x)$ contains the values of the breakpoints as well as the multilinear interpolation expressions in between the breakpoints of L.

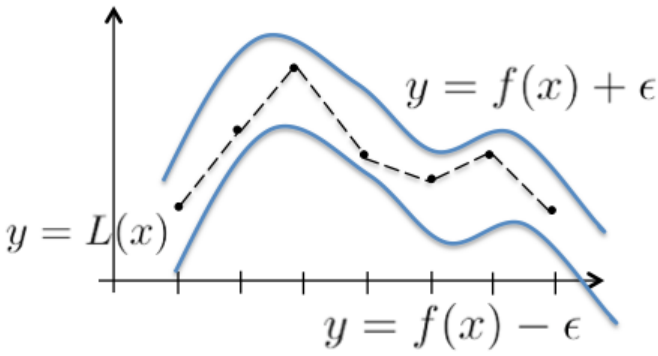

Figure 1: Lookup table function $L(x)$ abstracted by upper and lower bounding functions, obtained by shifting an approximation $f(a^*,x)$.

If the validity check fails, i.e. the SMT solver is able to find an $x \in S$ such that the lookup table produces a value outside of the upper and lower bounds, we try again with a larger value of ϵ. If it succeeds, with this value as the upper cap (valid ϵ) and 0 (invalid ϵ) as the lower cap we do a bisection search to find the smallest value of ϵ (within some tolerance) such that the offset functions abstract the lookup table. This yields a functional interval,

$$A(x) = [f(a^*,x) - \epsilon, f(a^*,x) + \epsilon]$$

such that for all $x \in S$, $L(x) \in A(x)$. This relationship is illustrated in Figure 1

5.2 Falsification

If the verification attempt does not succeed, it means that a value $x = \hat{x}$ was found such that the abstractions were satisfied, but the specification was falsified. This *candidate counterexample* is not necessarily a true counterexample, since a point that satisfies the abstractions may not satisfy the lookup tables.

However, this candidate counterexample serves as a flag of a region that may contain a true counterexample. It is sensible to search between the breakpoints that contain this counterexample, but note that this point may fall between different breakpoints in different lookup tables, which could potentially lead us to choose intervals from different lookup tables that are inconsistent with each other. To prevent this, instead of simply selecting the two breakpoints that contain the candidate counterexample, we select a small number r of the nearest breakpoints. See Figure 2 for an illustration of this mechanic. In our experiments, $r = 3$ or $r = 4$ are usually large enough to prevent inconsistent intervals.

Informally, we construct new lookup tables with only r entries each, and attempt to verify the same model with the reduced lookup tables, this time directly, without abstractions. If the verification succeeds, we know the candidate counterexample was spurious, and can repeat the procedure with a different candidate counterexample. If the verification fails, it provides a true counterexample which can be returned to the engineer as a design flaw that must be fixed.

Formally, let $x_j, \ldots, x_{j+n}$ be the n inputs of lookup table L_i. Then, consider the values of these variables in the candidate counterexample $\hat{x}_j, \ldots, \hat{x}_{j+n}$. We wish to extract the r nearest entries

along each dimension—suppose they are $x_j^{(k)}, \ldots, x_j^{(k+r)}$ through $x_{j+n}^{(k)}, \ldots, x_{j+n}^{(k+r)}$. Then, construct a new lookup table $\hat{L}_i$ that contains only these breakpoints, and maps them to the same outputs as L_i. Finally, check satisfiability of the following logical formula.

$$\left(\bigwedge_i \hat{L}_i\right) \wedge \Sigma(x) \wedge \neg S(x) \tag{2}$$

If a satisfying instance is found, then that instance is a true counterexample of the original model. If no satisfying instance is found, then we can try the procedure with a different candidate counterexample that is at some minimum distance δ from $\hat{x}$. To do this $\hat{x}_j, \ldots, \hat{x}_{j+n}$

If there are no more candidate counterexamples at this minimum distance, we move on to the next step, which is to refine the abstractions and attempt verification again.

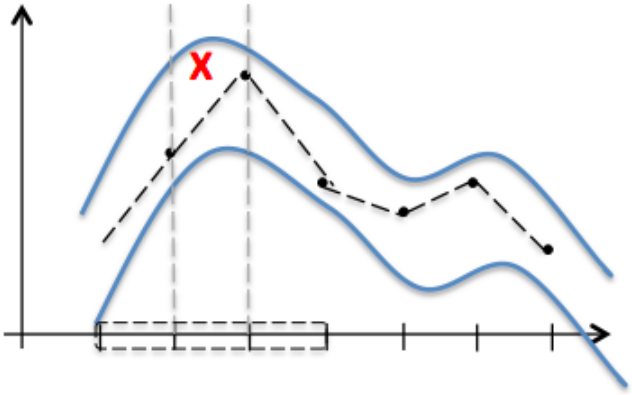

Figure 2: The red X represents a candidate counterexample. To search for a true counterexample, we construct a reduced table consisting of the four nearest breakpoints, which comprise the three intervals marked by the dotted rectangle

5.3 Abstraction refinement

When the SMT solver finds candidate counterexamples, meaning it is unable to prove correctness, and the falsification procedure fails to find a true counterexample, we refine the abstractions and repeat the verification attempt. There are two basic mechanisms by which we refine abstractions: (1) increasing arithmetic complexity of the templates, and (2) increasing the number of cases in a piecewise template.

Our tool implementation tries both of these techniques at the same time, and keeps the technique that yields the approximation with lowest error.

Increasing arithmetic complexity means moving from linear templates to quadratic templates, higher-order polynomials, or possibly transcendental functions if one is using an SMT solver that supports such functions, such as [8]. Increasing the number of cases in a piecewise template means moving from a simple equational template to a template with two cases, or from two to three, etc.

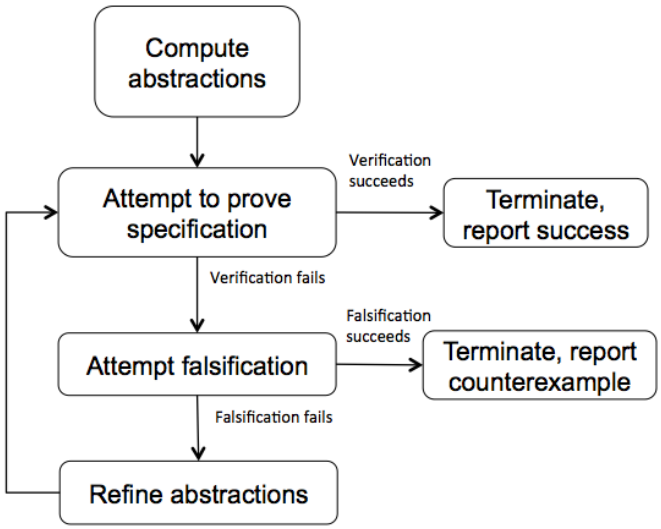

Figure 3: High-level view of Osiris

6 TOOL IMPLEMENTATION

We have implemented our technique in a tool called "Osiris". Figure 3 shows the overall control flow of the tool.

6.1 Input

The input to Osiris is a directory which contains

(1) a file with extension `*.fmlas`, which lists the bounds on the inputs of the model, as well as calculations by system elements that are not lookup tables, and
(2) for each lookup table, a *.m file, which contains a lookup table in standard Matlab syntax.

6.2 Learning abstractions with an extensible template library

Osiris reads the templates for the approximating functions from an external library, which can be modified and extended by the user. The templates are sorted by complexity, starting by linear templates, then quadratic templates, etc.

Osiris automatically starts working with a batch of the lowest complexity templates and computes approximations from them in parallel. All approximations are turned into abstractions by computing the smallest ϵ offset that produces upper and lower bounding function to the lookup table function. Checking the upper and lower bound properties is carried out with z3.

Since the learning procedure is not, for general templates, convex and therefore not guaranteed to converge to the same solution from different initial parameter values, it is advantageous to have multiple copies of the same template at the same complexity index in the library. In this way, Osiris solves multiple instances in parallel with different initial parameter values. The final selection for lowest ϵ ensures that we will be able to keep the best abstraction. Figure 4 shows this flow, where bold lines indicate multiple parallel instances.

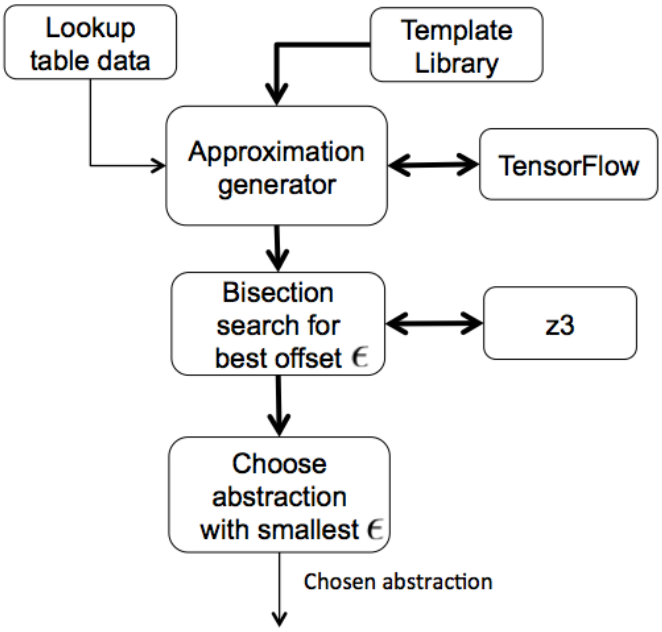

Figure 4: Flow of the abstraction generation procedure. Bold lines indicate multiple parallel instances.

Our library also has a section for domain-specific templates. We envision that future users of our tool will be interested in augmenting the library with templates from engine control, fluid dynamics, and other applications. These domain-specific templates allow computing abstractions that better match the mathematics of the application domain.

6.3 Learning abstractions

Osiris works through the template library in order of increasing complexity, using the machine learning tool TensorFlow to learn parameter values [1]. We chose to use TensorFlow to facilitate future extensions of our tool to more complex abstractions. Osiris uses the SMT solver z3 in the bisection search procedure to find the minimal offset ϵ that produces a true overapproximation of the lookup table function.

6.4 Proving specifications

Once each abstraction A_i has been generated for each lookup table L_i ($i = 1, \ldots, k$), Osiris forms the following logical formula.

$$A_1(x) \wedge \cdots \wedge A_k(x) \wedge \Sigma(x) \wedge \neg S(x)$$

Then, Osiris invokes the SMT solver z3 to check for satisfiability. If the formula is not satisfiable, z3 has proven that there is no value that satisfies the abstractions and the model constraints but falsifies the specification. Since the abstractions overapproximate the lookup table functions, it follows that the system with the lookup table functions satisfies its specifications. If the formula was satisfiable, Osiris proceeds to the falsification stage.

6.5 Falsification

If a violation of the safety property $\hat{x}$ is found, this does not necessarily mean that the original system violates its specifications. For each lookup table L_i, Osiris finds nearest breakpoints in each lookup table. Then, Osiris tries to prove the correctness of the model *only between those breakpoints*. If the verification fails, the result is now a true counterexample, which can be reported to the designer. If no true counterexample is found, Osiris tries to compute new abstractions with the next set of templates in the template library.

7 CASE STUDY

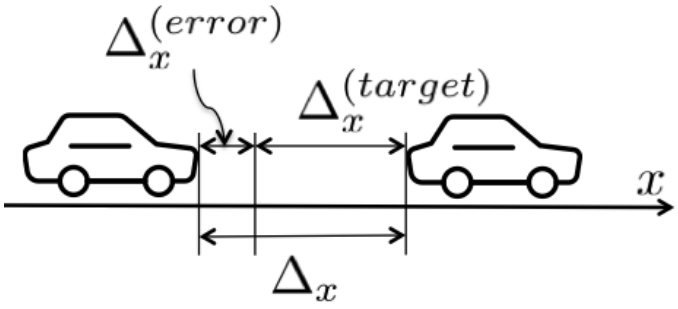

Figure 5: Diagram of adaptive cruise control scenario

For our case study, we consider the verification benchmark published by Toyota InfoTechnology Center at [16]. This benchmark consists of an adaptive cruise controller along with an online monitor. When enabled, adaptive cruise control regulates the speed of the car so that a target speed is maintained, unless another car is detected at some distance in front, in which case the system tries to maintain a safe distance from the lead car, as shown in Figure 5. This controller takes as input the current speed of the car, the distance to the lead car, and the relative speed between the two cars.

The system consists of a cascade of three lookup tables, as shown in Figure 6. The inputs to the controller are s, the speed of the controlled car, Δ_x, the distance to the leading car, and Δ_v the relative speed of the two cars.

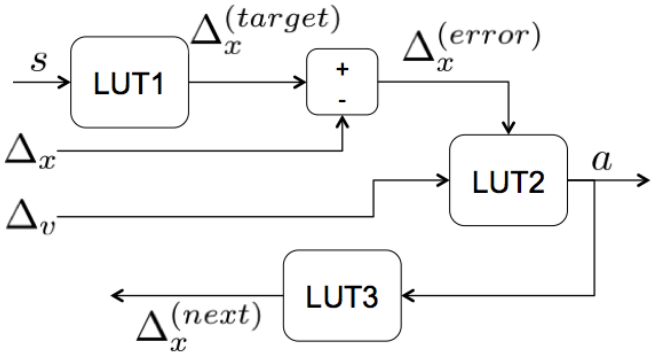

Figure 6: Signal-flow model of an ACC controller

The first lookup table uses the current speed s of the controlled car to determine a target set distance ($\Delta_x^{(target)}$) from the leading car. If the controlled car is moving fast, its braking distance will be larger, which requires the controller to choose a larger following distance. $\Delta_x^{(error)}$ is the difference between the target following distance and the chosen following distance, and the second lookup table uses $\Delta_x^{(error)}$ together with the relative velocity Δ_v to choose an acceleration a. The third lookup table behaves as an online monitor. In practice, a monitor lookup table would be produced by recording observations of a physical component. For this example, the monitor was generated by computing the future distance between the two cars after 0.1 seconds, given the current distance, relative velocity, and chosen acceleration. This monitor assumes that the lead car will not change its velocity within the next 0.1 seconds.

The property we wish to prove is that the online monitor will never predict a future distance that is negative, i.e., it will never predict that the cars will crash. This does not mean that the closed-loop system with the real automotive dynamics will not crash, since that would require analyzing the continuous-time differential equations. However, industrial controllers are frequently equipped with online monitors that predict or prevent dangerous conditions, and checking that the controller satisfies its monitor is valuable, as it prevents any abnormal behavior as long as system integrity is preserved. However, Osiris is not limited to analyzing properties based on a monitor, and can analyze general properties expressed in first-order logic over the variables of the model.

The first lookup table is one-dimensional and contains 21 breakpoints, resulting in 22 possible interval values. The second lookup table is two-dimensional, and has a total of $1,232$ possible interval values. The third lookup table is three-dimensional and has $2,856$ possible values. A brute-force attempt at proving correctness would need to consider all possible combinations of lookup table values. Considering all possible cominations of internal values leads to a total of $77,409,024$ proof cases.

We translated our model to first order logic to use an SMT solver to check validity of the specification. We do not explicitly include the logical formulas that represent the lookup table due to space constraints, but they can be easily constructed by multilinear interpolation on the public benchmark files. The translated first order logic constraints are:

$$
\begin{aligned}
&0 \le \Delta_x \le 180,\\
&-50 \le \Delta_v \le 50,\\
&0 \le s \le 180,\\
&\Delta_x^{(target)} = \mathsf{LUT}_1(s),\\
&\Delta_x^{(error)} = \Delta_x - \Delta_x^{(target)},\\
&a = \mathsf{LUT}_2(\Delta_x^{(error)}, \Delta_v),\\
&\Delta_x^{(next)} = \mathsf{LUT}_3(\Delta_x, \Delta_v, a).
\end{aligned}
$$

The constraints on Δ_x, Δ_v, and s are assumptions on the bounds of these inputs, and the system cannot be enabled if these bounds are not met. Similarly, commercial adaptive cruise control systems cannot be used if the speed of the controlled car is too slow.

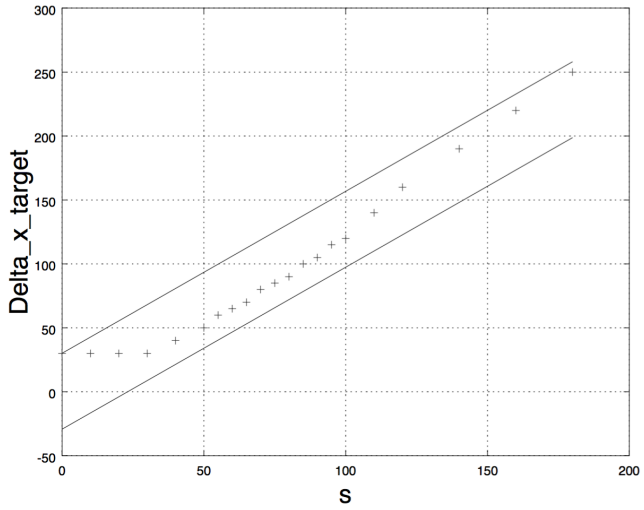

Figure 7: Plot of LUT 1 data and abstractions

We tried to verify the model directly by translating it into first-order logic constraints and using z3 to check for a violation of the specification, but z3 did not terminate after 48 hours.

When we ran this model in Osiris, a counterexample was found in 1 minute and 50 seconds, as follows:

$$
\begin{aligned}
&s \mapsto 31.0,\\
&a \mapsto -2.0,\\
&\Delta_v \mapsto -4.0,\\
&\Delta_x \mapsto 0.03125,\\
&\Delta_x^{(error)} \mapsto -30.97,\\
&\Delta_x^{(target)} \mapsto 31.0,\\
&\Delta_x^{(next)} \mapsto -0.00865.
\end{aligned}
$$

The meaning of this counterexample is that the cars start at a distance Δ_x of about 3 cm, with a relative velocity of $-4m/s$, i.e. the controlled car is moving $4m/s$ faster than the lead car. The controller brakes by applying a negative acceleration of $a = -2m/s^2$, but the situation is already too dangerous and the cars have a minor crash, with the controlled car being $0.8cm$ further than it should be.

To measure the runtime of our verification technique, we relaxed the specification to $\Delta_x^{(next)} \ge -2$. With this relaxed property, the monitor no longer tries to completely prevent collisions, but simply to reduce their severity. Of course, this is not a controller that could be deployed for a commercial automotive system, it is simply for benchmarking of our tool. This relaxed property was provable in 30 seconds, which compares favorably with an analysis time of approximately four hours in [11] on a machine with the same specifications. The case study computations were carried out on a machine with 44 cores, with available hyperthreading to 88 threads and 256 GB of RAM.

Computed abstractions. The abstraction computed for LUT1 consists of a linear function, shifted above and below the lookup table

data.

$$A_1 = [1.31s - 4.0315 - \epsilon_1, 1.31s - 4.0315 + \epsilon_1]$$

We have deliberately left the constants un-simplified. The constant $\epsilon_1 = 34.1797$ is useful because it represents the largest error between the abstraction and the lookup table itself. Thus, we can compare which lookup tables are being abstracted with more or less fidelity by looking at the value of ϵ.

The abstraction computed for LUT2 is a linear function, and has the form

$$A_2 = [f_2 - \epsilon_2, f_2 + \epsilon_2]$$

where

$$f_2 = 0.023843553\Delta_x^{(error)} + 0.091889\Delta_v - 0.51779$$

with $\epsilon_2 = 3.90625$.

The abstraction computed for LUT3 is a linear function.

$$A_3 = [f_3 - \epsilon_3, f_3 + \epsilon_3]$$

where

$$f_3 = 0.99876517\Delta_x + 0.00795821\Delta_v - 0.0016369a$$

and $\epsilon_3 = 0.5859375$.

8 CONCLUSION

We have developed a tool to automatically compute abstractions of lookup tables. We treat the lookup table breakpoints as training data for a regression procedure, and learn an abstraction of the lookup table. Abstracting the lookup tables allows for fast, automatic verification of input-output properties of large-scale controllers with lookup tables.

Osiris parses a set of logical constraints along with lookup tables represented as Matlab programs, as well as regression templates from an extensible library, which is sorted by complexity. Starting at the lowest level of complexity, Osiris attempts to generate abstractions to prove the desired specification, increasing the level of complexity after each failed attempt. The extensible nature of our template library makes it easy for engineers in different application domains to add templates that may provide a good fit to the lookup tables that appear in their discipline.

We have demonstrated the effectiveness of our approach on a public benchmark, which consists of an adaptive cruise controller with lookup tables. The best published analysis time for this model is around four hours [11], and Osiris has completed the analysis in thirty seconds on a machine with the same specifications.

9 FUTURE WORK

We have identified several directions for future work. We would like to generalize the form of the abstractions that we use, since our current abstractions are limited to upper and lower bounding functions. Also, we would like to explore the possibility of using a learning method with better theoretical guarantees (e.g. SVMs), since our current setup yields a non-convex optimization problem.

The current implementation of Osiris can only check single specifications. In the case when the specification does not hold, one can obtain insight by relaxing the specification. In future work, we would like to support sets of specifications, and to construct a lattice of relaxed specifications, which Osiris would iteratively attempt to prove. In this way, we would be able to provide the designer with a trade-off curve of which of the desired specifications are easier or more difficult to prove. We also plan to extend our template library as we gain experience with further case studies.

Osiris currently only supports static SMT tools as back-end solvers. In future work, we would like to extend the applicability of our technique to handle closed-loop control systems, which would require a hybrid model checking tool, or some kind of automatic invariant guessing heuristic.

Additionally, we would like to provide better feedback to the designer about which parts of the design seem to be most critical to satisfying the specifications. This would require an efficient way to quantify the expected improvement in the specification with respect to the improvement of each abstraction.

REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Josefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. 2015. *TensorFlow.* Technical Report. Google Research.

[2] Aditya Agrawal, Gyula Simon, and Gabor Karsai. 2004. Semantic Translation of Simulink/Stateflow Model to Hybrid Automata Using Graph Transformations. *Electronic Notes in Theoretical Computer Science* (2004).

[3] Sergio E. Zarantonello Alan Weiser. 1988. A Note on Piecewise Linear and Multilinear Table Interpolation in Many Dimensions. *Math. Comp.* 50, 181 (1988), 189–196. http://www.jstor.org/stable/2007922

[4] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided Abstraction Refinement. (2000).

[5] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (International Conference on Tools and Algorithms for the Construction and Analysis of Systems'08/ETAPS'08).* Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[6] A. Donzé. 2010. Breach, A TooTool for Verification and Parameter Synthesis of Hybrid Systems. In *Computer Aided Verification.*

[7] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. 2011. SpaceEx: Scalable verification of hybrid systems. In *Int. Conf on Computer Aided Verification.* 379–395.

[8] S. Gao, S. Kong, and E. M. Clarke. 2013. dReal: An SMT Solver for Nonlinear Theories of the Reals. In *Int. Conf. on Automated Deduction.*

[9] S. Gao, S. Kong, and E. M. Clarke. 2013. Satisfiability Modulo ODEs. In *Formal Methods in Computer-Aided Design (FMCAD), 2013.* IEEE, 105–112.

[10] D. Harel, D. Kozen, and J. Tiuryn. 2000. *Dynamic Logic.* MIT Press, Foundations of Computing Series.

[11] A. B. Hocking, M. A. Aiello, J. C. Knight, and N. Aréchiga. 2017. Input Space Partitioning to Enable Massively Parallel Proof. In *NASA Formal Methods Symposium.*

[12] Jean-Baptiste Jeannin, Khalil Ghorbal, Yanni Kouskoulas, Ryan Gardner, Aurora Schmidt, Erik Zawadzki, and André Platzer. 2015. Formal Verification of ACAS X, an Industrial Airborne Collision Avoidance System. In *EMSOFT*, Alain Girault and Nan Guan (Eds.). IEEE Press, 127–136. DOI: http://dx.doi.org/10.1109/EMSOFT.2015.7318268

[13] X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts. 2014. Benchmarks for Model Transformations and Conformance Checking. In *Workshop on Applied Verification for Continuous and Hybrid Systems.*

[14] J. Kapinski, J. Deshmukh, S. Sankaranarayanan, and N. Aréchiga. 2014. Simulation-guided Lyapunov Analysis for Hybrid Dynamical Systems. In *Int. Conf. on Hybrid Systems: Computation and Control.*

[15] Ashish Tiwari. 2002. *Formal Semantics and Analysis Methods for Simulink Stateflow Models.* Technical Report. SRI International.

[16] M. Yamaura, N. Aréchiga, and S. Shiraishi. 2017. Simulink Verification Benchmark. https://github.com/Toyota-ITC-SSD/SimulinkVerificationBenchmark. (2017).